HEAP

Structura de tip *Heap*

Heap-ul, figura 15.1, este un arbore binar care respectă proprietăţile de structură şi de ordonare. Fiecare nod al arborelui trebuie să conţină o valoare asociată numită cheie şi poate conţine alte informaţii suplimentare. Cheia trebuie să permită definirea unei relaţii de ordine totală pe mulţimea nodurilor. În funcţie de obiectivul urmărit, heap-ul poate fi organizat sub formă de *max-heap* sau *min-heap*. Cele două tipuri de heap sunt echivalente. Transformarea unui max-heap în min-heap, sau invers, se poate realiza prin simpla inversare a relaţiei de ordine.

32

24 **8**

12 **23** **2** **3**

7 9 17

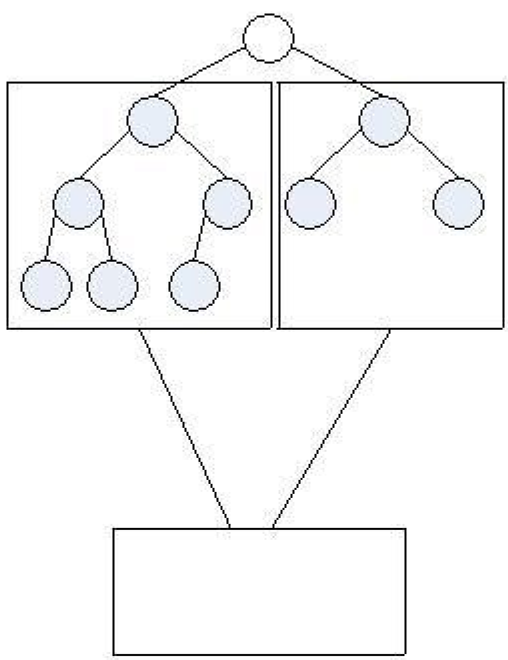
Figura 1 *Reprezentarea grafică a structurii heap*

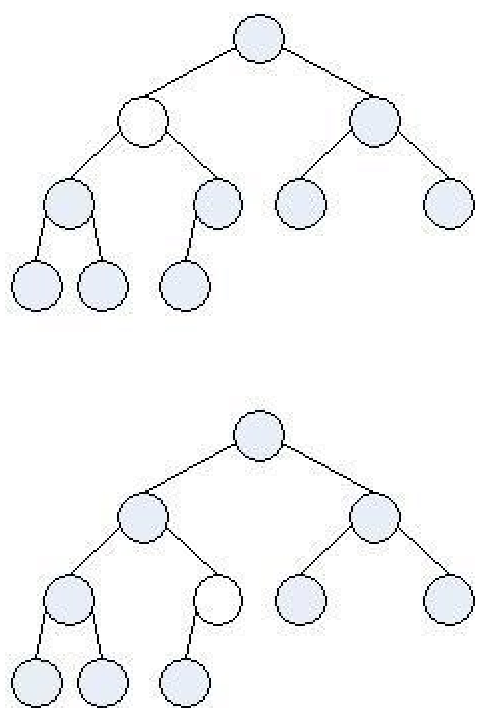
Proprietatea de structură specifică faptul că elementele sunt organizate sub forma unui arbore binar complet. Prin arbore binar complet înţelegem un arbore binar în care toate nodurile, cu excepţia celor de pe ultimul nivel, au exact doi fii, iar nodurile de pe ultimul nivel sunt completate de la stânga la dreapta.

Proprietatea de ordonare impune ca valoarea asociată fiecărui nod, cu excepţia nodului rădăcină, să fie mai mică sau egală decât valoarea asociată nodului părinte. Se observă că, spre deosebire de arborii binari de căutare, nu se impune nici o regulă referitoare la poziţia sau relaţia dintre nodurile fiu.

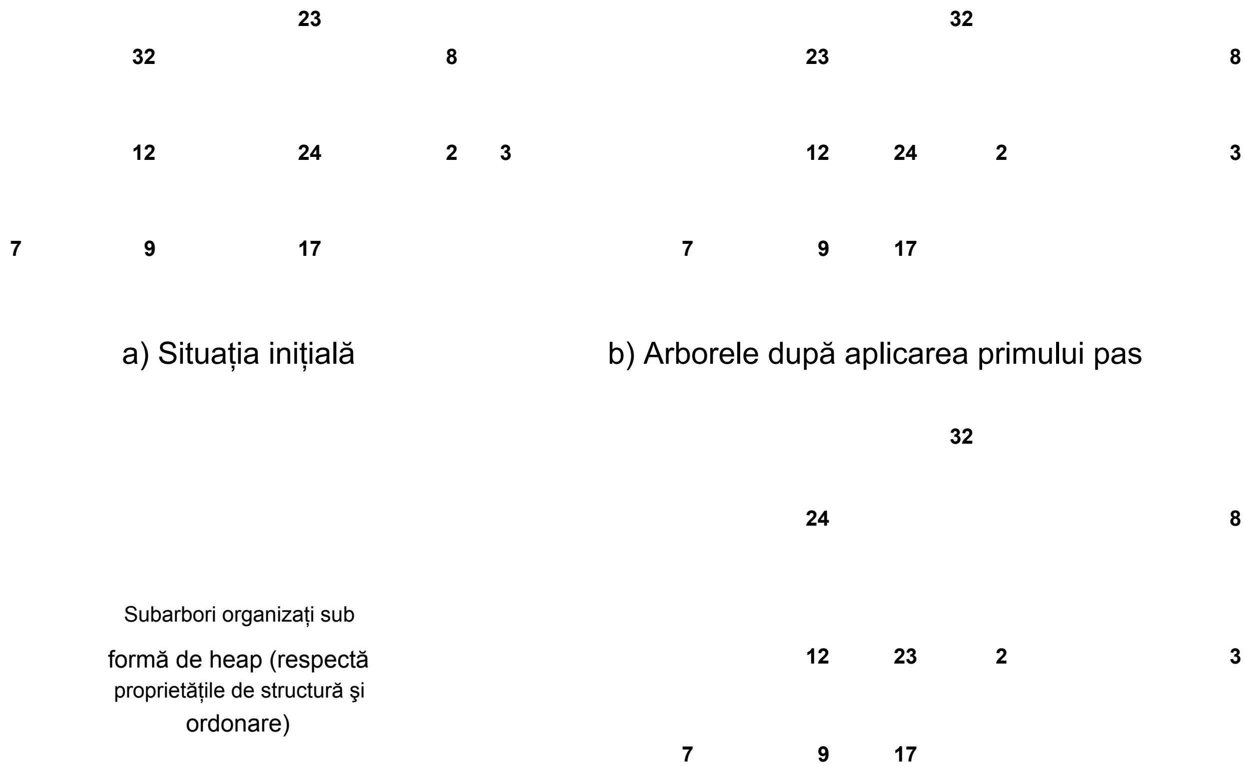
Structura heap este preferată pentru multe tipuri de aplicaţii. Cele mai importante utilizări sunt:
implementarea cozilor de prioritate utilizate pentru simularea pe bază de evenimente sau algoritmi de alocare a resurselor;
implementarea selecţiei în algoritmi de tip greedy cum ar fi algoritmul lui Prim pentru determinarea arborelui de acoperire minimă sau algoritmul lui Dijkstra pentru determinarea drumului minim;
sortarea masivelor utilizând algoritmul HeapSort.

Operaţiile principale care se execută pe o structură heap sunt cele de construire a heap-ului pornind de la un masiv unidimensional oarecare, inserarea unui element în structură şi extragerea elementului maxim sau minim pentru un min-heap.





Construirea structurii se face utilizând o procedură ajutătoare numită procedură de filtrare. Rolul acesteia este de a transforma un arbore în care doar subarborii rădăcinii sunt heap-uri ale căror înălţime diferă cu cel mult o unitate într-un heap prin coborârea valorii din rădăcină pe poziţia corectă. Structura rezultată în urma aplicării procedurii de filtrare este un heap (respectă proprietăţile de structură şi ordonare).



a) Situația inițială

b) Arborele după aplicarea primului pas

c) Arborele la sfârşitul procedurii de filtrare

Figura 2 *Aplicarea algoritmului de filtrare*

Algoritmul de filtrare, figura 15.2, presupune parcurgerea următoarelor etape începând cu nodul rădăcină:

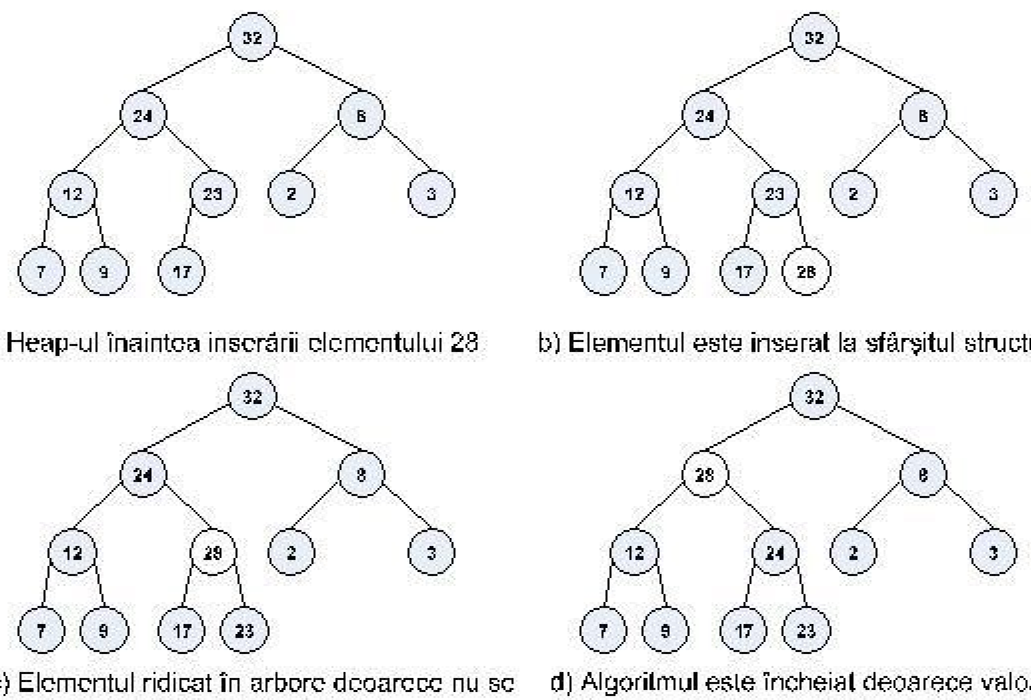
se determină maximul dintre nodul curent, fiul stânga şi fiul dreapta (dacă există).

dacă maximul se află în nodul curent, atunci algoritmul se opreşte.

dacă maximul de află într-unul dintre fii, atunci se interschimbă valoarea din nodul curent cu cea din fiu şi se continuă execuţia algoritmului cu nodul fiu.

Construirea heap-ului pornind de la un arbore binar care respectă doar proprietatea de structură se face aplicând procedura de filtrare pe nodurile non frunză ale arborelui începând cu nodurile de la baza arborelui şi continuând în sus până când se ajunge la nodul rădăcină. Corectitudinea algoritmului este garantată de faptul că la fiecare pas subarborii nodului curent sunt heap-uri (deoarece sunt noduri frunză sau sunt noduri pe care a fost aplicată procedura de filtrare).

Inserarea elementelor într-un heap se poate face şi după etapa iniţială de construcţie. În acest caz adăugarea elementului nou trebuie făcută astfel încât structura rezultată să păstreze proprietatea de ordonare. Inserarea unui element, figura 15.3, în heap presupune parcurgerea următoarelor etape:



a) Heap-ul înaintea inserării elementului 28

b) Elementul este inserat la sfârşitul structurii

c) Elementul ridicat în arbore deoarece nu se respectă proprietatea de ordonare

d) Algoritmul este încheiat deoarece valoarea nodului inserat este mai mică decât valoarea nodului părinte

se adaugă elementul ca nod frunză la sfârşitul arborelui pentru a păstra proprietatea de structură;
se compară nodul curent cu nodul părinte;

dacă nodul părinte este mai mic se interschimbă nodul curent cu nodul părinte;
dacă nodul părinte este mai mare sau egal atunci algoritmul se opreşte.

Figura 3 *Aplicarea algoritmului de inserare*

Procedura prezentată permite inserarea rapidă a oricărei valori în cadrul heap-ului cu păstrarea proprietăţilor de structură şi de ordonare.

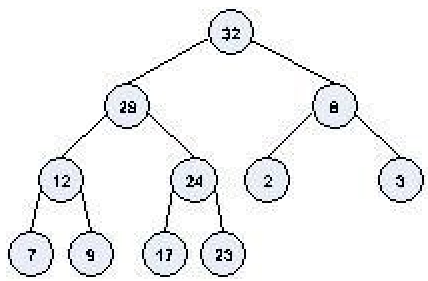
Ştergerea elementelor dintr-un heap se poate efectua doar prin extragerea elementului maxim sau minim în cazul unui min-heap. Pentru păstrarea structurii de heap se utilizează procedura de filtrare prezentată anterior. Algoritmul de extragere al elementului maxim, figura 15.4, presupune parcurgerea următoarelor etape:

se interschimbă valoarea din nodul rădăcină cu valoarea din ultimul nod al arborelui;

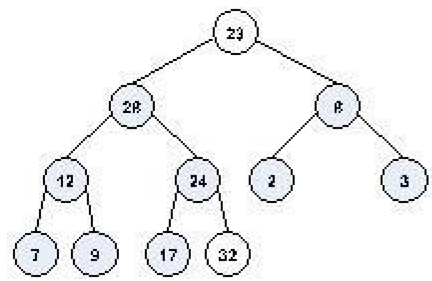
se elimină ultimul nod din arbore;

se aplica procedura de filtrare pe nodul rădăcină pentru a păstra proprietatea de ordonare;

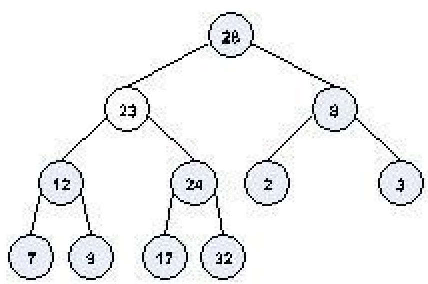
se returnează valoarea din nodul eliminat.



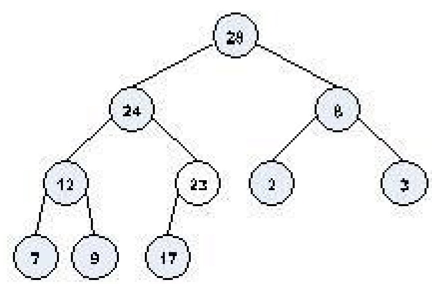
a) Heap-ul înaintea extragerii elementului maxim



b) Se interschimbă rădăcina cu ultimul nod



c) Se aplică procedura de filtrare pentru coborârea nodului pe poziția corectă



d) După încheierea procedurii de filtrare se elimină ultimul nod din structură

Figura 4 *Aplicarea algoritmului extragere element maxim*

Operaţiile prezentate în secţiunea 15.1 permit utilizarea structurii heap ca punct de plecare pentru implementarea eficientă a cozilor de prioritate şi a algoritmului de sortare HeapSort.

15.2 Implementarea structurii *Heap*

Deşi este posibilă implementarea structurii heap folosind arbori binari, datorită particularităţilor arborelui de tip heap, stocarea eficientă a acestuia poate realizează fără ajutorul pointerilor, folosind un masiv unidimensional. Elementele arborelui se stochează în masiv începând cu nodul rădăcină şi continuând cu nodurile de pe nivelurile următoare preluate de la stânga la dreapta.

Reprezentarea sub formă de masiv a heap-ului din figura 15.1 este prezentată în figura 15.5.

32	24	8	12	23	2	3	7	9	17
0	1	2	3	4	5	6	7	8	9

Figura 15.5 *Reprezentarea în memorie pentru structura heap*

Navigarea între elementele arborelui se poate face în ambele direcţii folosind următoarele formule:

$$\text{Parinte}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \ \text{Stânga}(i) = 2i+1, \ \text{Dreapta}(i) = 2i+2 \qquad (15.1)$$

Pentru implementarea structurii heap în limbajul C++ a fost construită o clasă template denumită *Heap* care implementează algoritmii prezentaţi anterior. Clasa permite construirea de heap-uri pentru orice tip de date care implementează operatorul „<" (necesar pentru construirea relaţiei de ordine pe mulţimea nodurilor) şi un constructor implicit.

Interfaţa clasei este următoarea:

Se observă că, în afara datelor efective stocate în masiv, este necesară stocarea a două informaţii suplimentare:
dimensiunea memoriei alocate pentru structură;
numărul de elemente prezente efectiv în structură.

Memoria necesară stocării masivului este alocată dinamic şi gestionată automat de către clasa Heap prin intermediul metodelor

```
{
//alocăm un vector nou mai mic şi copiem elementele T* masivNou =
new T[this->memorieAlocata / 2];
for (int i = 0; i < this->memorieAlocata / 2; i++)
{
masivNou[i] = this->elemente[i];
}
```

```
//înlocuim vectorul existent cu cel nou şi actualizăm dim delete
[] this->elemente;
this->elemente = masivNou;
this->memorieAlocata = this->memorieAlocata / 2;
}
```

Memoria alocată de către instanţele clasei este dealocată de către destructor. Pentru a evita cazul în care două instanţe ale clasei *Heap* conţin referinţe către aceleaşi elemente a fost interzisă operaţia de atribuire prin supraîncărcarea privată a operatorului corespunzător.

Pentru construirea structurii heap pe baza unui masiv oarecare şi pentru asigurarea proprietăţii de ordonare în cazul metodei de extragere maxim a fost implementată metoda ajutătoare *Filtrare* conform algoritmului prezentat în secţiunea 15.1: Implementarea constructorilor în cadrul clasei *Heap* este următoarea:

3 Cozi de prioritate

Cozile de prioritate sunt structuri de date care suportă următoarele două operaţii de bază:
inserarea unui element cu o prioritate asociată;
extragerea elementului cu prioritate maximă.

Cele mai importante aplicaţii ale cozilor de prioritate sunt: simularea bazată pe evenimente, gestionarea resurselor partajate (lăţime de bandă, timp de procesare) şi căutare în spaţiul soluţiilor (de exemplu, algoritmul A* utilizează o coadă de prioritate pentru a reţine rutele neexplorate).

Structura de date de tip Heap este una dintre cele mai eficiente modalităţi de implementare a cozilor de prioritate. Prioritatea elementelor este dată de relaţia de ordine existentă între valorile asociate nodurilor. Pentru exemplificarea modului de utilizare a clasei Heap prezentată în secţiunea 15.2 vom construi un simulator discret pentru o coadă de aşteptare la un magazin.

În simularea discretă, modul de operare al unui sistem este reprezentat sub forma unei secvenţe de evenimente ordonate cronologic. În cazul de faţă evenimentele sunt sosirile clienţilor în coada de aşteptare şi servirea clienţilor. Simulatorul conţine o coadă de evenimente. Evenimentele sunt adăugate în coadă pe măsură ce timpul lor de producere poate fi determinat şi sunt extrase din coadă pentru procesare în ordine cronologică.

Un simulator discret pe bază de evenimente are următoarele componente:
coada de evenimente – o coadă de prioritate care conţine lista evenimentelor care se vor petrece în viitor;
starea simulatorului – conţine un contor pentru memorarea timpului curent, informaţiile referitoare la starea actuală a sistemului simulat (în cazul curent clienţii aflaţi în coadă şi starea staţiei de servire) şi indicatori;
logica de procesare – extrage din coadă evenimentele în ordine cronologică şi le procesează; procesarea unui eveniment determină modificarea stării sistemului şi generarea de alte
evenimente.

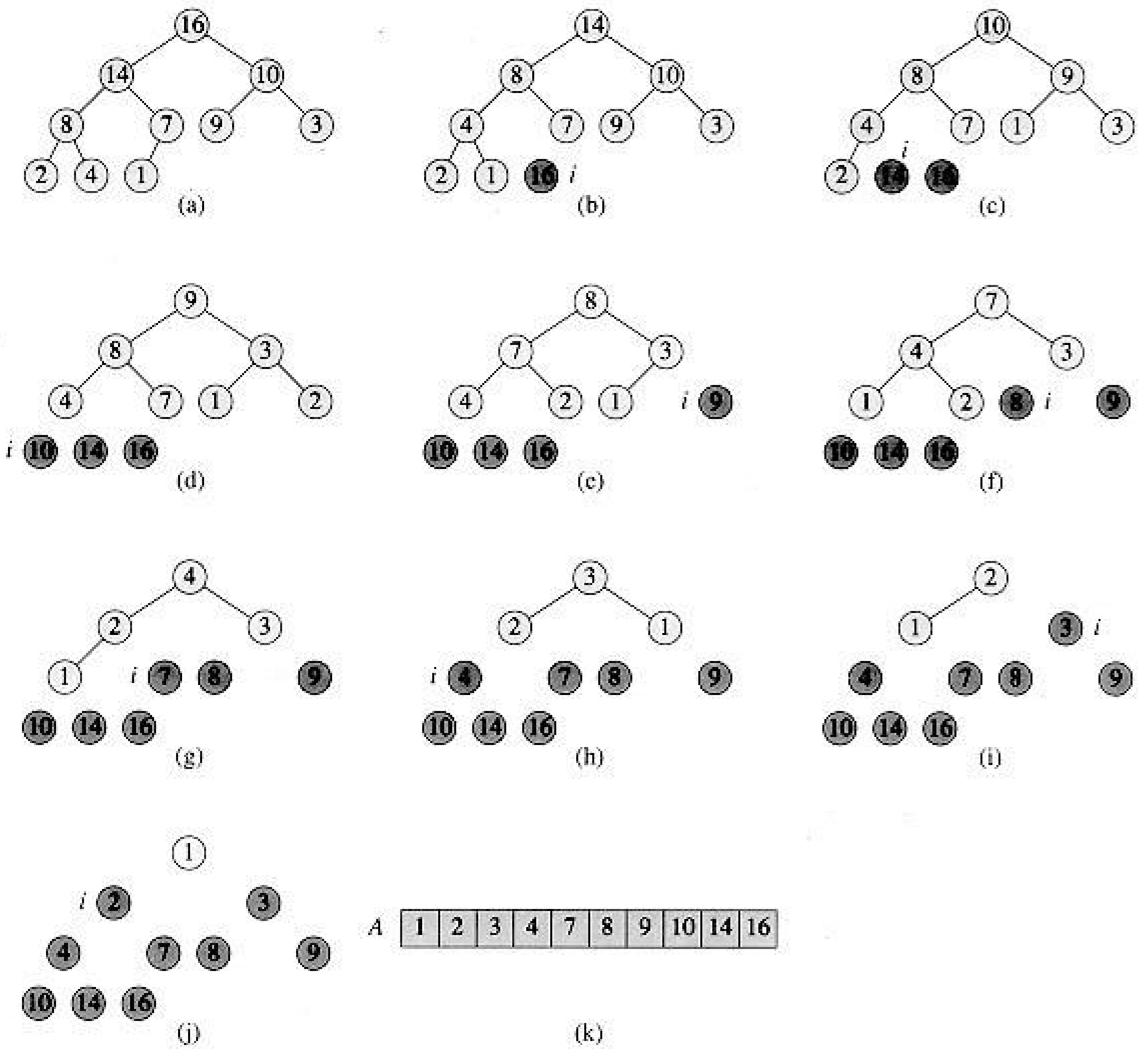
Pentru simularea propusă au fost luate în considerare următoarele ipoteze:
există o singură staţie de servire cu un timp de servire distribuit normal, cu o medie şi dispersie cunoscută;
există o singură coadă pentru clienţi, iar intervalul de timp dintre două sosiri este distribuit uniform într-un interval dat;
durata simulării este stabilită de către utilizator.

Simularea se realizează prin extragerea evenimentelor din heap şi procesarea acestora pe bază de reguli. Evenimentele de tip sosire determină generarea evenimentului corespunzător sosirii următoare şi a unui eveniment de servire în cazul în care staţia este liberă la momentul curent. În cazul evenimentelor de tip servire se generează următorul eveniment de tip servire dacă mai există clienţi în coadă. Pe măsură ce sunt procesate evenimentele sunt reţinute şi informaţiile necesare pentru calcularea indicatorilor de performanţă aferenţi sistemului simulat.

Aspecte teoretice:
Un ”Heap”, numit uneori ansamblu sau movilă este un tablou de elemente H[1..n] care are proprietatea ca pentru fiecare element H[i] cu 1<=i<=[n/2] sunt adevărate inegalitățile: H[i]>=H[2i] si H[i]>=H[2i+1] (în cazul în care 2i+1<=n. O astfel de structură poate fi vizualizată sub forma unui arbore binar aproape complet (arbore în care fiecare nod are doi fii, iar fiecare nivel cu excepția ultimului este complet). fii unui nod aflat pe poziția i în tablou se află pe pozițiile 2i (fiul stâng) respectiv 2i+1 (fiu drept). În baza aceleiași proprietăți părintele nodului de pe poziția i în tablou se află pe poziția [i/2].
Aceste structuri de date sunt utile atât pentru sortarea eficientă a unui tablou cât și pentru implementarea cozilor de priorități. În această secțiune structura de tip “heap” este folosită doar pentru implementarea unei metode de sortare.
Procesul propriu zis de sortare constă în doua etape principale:

- Construirea, pornind de la tabloul inițial al unui heap. Procesul de construire se bazează pe utilizarea algoritmului reHeap pentru fiecare element din prima jumătate a tabloului începând cu elementul de pe poziția [n/2]
- Eliminarea succesivă din heap a nodului rădăcină și plasarea acestuia la sfârșitul tabloului corespunzător heap-ului. La fiecare etapă dimensiunea heap-ului scade cu un element iar subtabloul ordonat de la sfârșitul zonei corespunzătoare tabloului inițial crește cu un element.

Analiza complexității:
Întrucât algoritmul heapSort apelează de n-1 ori algoritmul reHeap care are ordinul de complexitate O(log n) iar cum algoritmul de construire are ordinul O(nlog n) rezultă ca heapSort are de asemenea ordinul O(nlog n).
Exemplu Pas cu Pas:


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Cod Sursa C++

```cpp
// HeapSortCPP.cpp : Defines the entry point for the console application.

//



#include "stdafx.h"

#include <iostream>



using namespace std;



int v[100];



void heapify(int v[], int root, int bottom)

{

	int done, maxChild, aux;

	done=0;

	while ((root*2 <= bottom) && (!done))

	{

		if(root*2 == bottom)

			maxChild = root *2;
```

```
        else
            if(v[root*2] > v[root*2+1])
                maxChild=root*2;
            else
                maxChild=root*2+1;
        if(v[root]<v[maxChild])
        {
            aux=v[root];
            v[root]=v[maxChild];
            v[maxChild]=aux;
            root=maxChild;
        }
        else
            done=1;
    }
}

void HeapSort(int v[], int len)
{
```

```
        int i,aux;

        for (i=(len/2)-1; i>=0; i--)

                heapify(v,i,len);

        for (i=len-1; i>=1; i--)

        {

                aux=v[0];

                v[0]=v[i];

                v[i]=aux;

                heapify(v,0,i-1);

        }

}


int main(void)

{

        int i;

        int len;

        cout<<"Dati numarul de elemente din vector: ";

        cin>>len;

        for (i=0; i<len; i++)
```

```
	{
		cout<<"v["<<i<<"]=";
		cin>>v[i];
	}
	HeapSort(v, len);
	for (i=0; i<len; i++)
		cout<<v[i]<<" ";
	system("pause");
}
```

Metode de sortare

Heapsort

Prin algoritmul heapsort se ordonează elementele în spaţiul alocat vectorului: la un moment dat doar un număr constant de elemente ale vectorului sunt păstrate în afara spaţiului alocat vectorului de intrare. Astfel, algoritmul heapsort combină calităţile a două tipuri de algoritmi de sortare, sortare internă şi sortare externă.

Heapsort introduce o tehnica nouă de proiectare a algoritmilor bazată pe utilizarea unei structuri de date, numită de regula *heap*.

Structura de date heap este utilă nu doar pentru algoritmul heapsort, ea poate fi la fel de utilă şi în tratarea eficientă a unei cozi de prioritate. Termenul heap a fost introdus şi utilizat iniţial în contextul algoritmului heapsort, dar acesta se foloseşte şi în legătură cu alocarea dinamică, respectiv în tratarea memoriei bazate pe "colectarea reziduurilor" (*garbage collected storage*), de exemplu în limbajele de tip Lisp.
Structura de date heap *nu* se refera la heap-ul menţionat în alocarea dinamica, şi ori de câte ori, în aceasta lucrare voi vorbi despre heap, vom înţelege structura definită aici pentru heapsort.
Structura de date *heap (binar)* este un vector care poate fi vizualizat sub forma unui arbore binar aproape complet, conform figurii 1.8.

Fiecare nod al arborelui corespunde unui element al vectorului care conţine valorile ataşate nodurilor.

Arborele este plin, exceptând eventual nivelul inferior, care este plin de la stânga la dreapta doar până la un anumit loc. Un vector *A* care reprezintă un heap are doua atribute: *lungime*[*A*], reprezintă numărul elementelor din vector şi *dimensiune-heap*[*A*] reprezintă numărul elementelor heap-ului memorat în vectorul *A*.

Astfel, chiar daca *A*[1..*lungime*[*A*]] conţine în fiecare element al sau date valide, este posibil ca elementele următoare elementului *A*[*dimensiune-heap*[*A*]], unde *dimensiune- heap*[*A*] ≤ *lungime*[*A*], să nu aparţină heap-ului. Rădăcina arborelui este *A*[1]. Dat fiind un indice *i*, corespunzător unui nod, se pot determina uşor indicii părintelui acestuia Parinte(*i*), al fiului Stânga(*i*) şi al fiului Dreapta(*i*).

Parinte(*i*)
    returneaza [*i*/2]
Stânga(*i*)
    returneaza 2*i*
Dreapta(*i*)
    returneaza 2*i* + 1

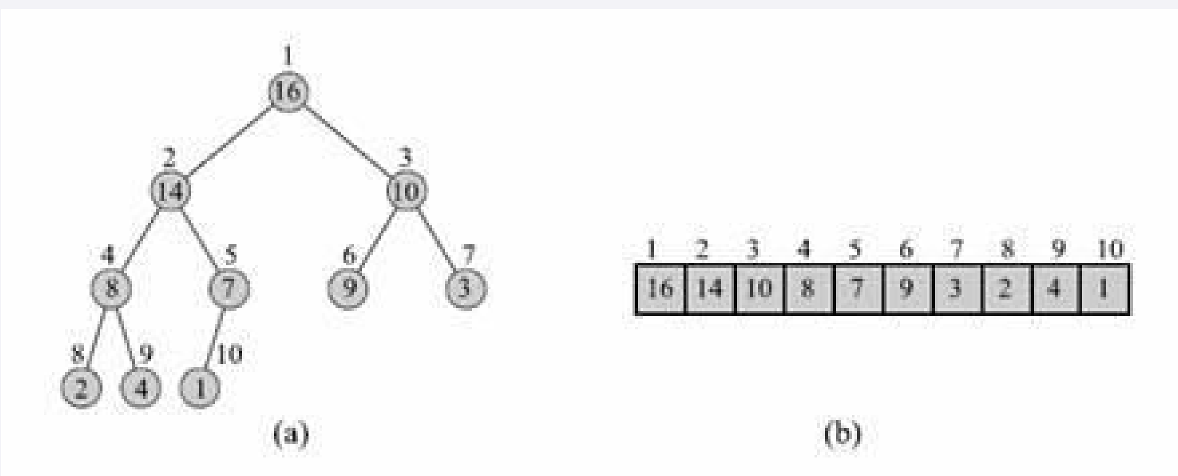


Figura 1.8

Un heap reprezentat sub forma unui arbore binar (a) şi sub forma unui vector (b). Numerele înscrise în cercurile reprezentând nodurile arborelui sunt valorile ataşate nodurilor, iar cele scrise lângă cercuri sunt indicii elementelor corespunzătoare din vector.

În cele mai multe cazuri, procedura Stânga poate calcula valoarea $2i$ cu o singura instrucţiune, translatând reprezentarea binara a lui i la stânga cu o poziţie binara. Similar, procedura Dreapta poate determina rapid valoarea $2i + 1$, translatând reprezentarea binara a lui i la stânga cu o poziţie binara, iar bitul nou intrat pe poziţia binara cea mai nesemnificativa va fi 1. În procedura Parinte valoarea $[i/2]$se va calcula prin translatarea cu o poziţie binara la dreapta a reprezentării binare a lui i. Într-o implementare eficienta a algoritmului heapsort, aceste proceduri sunt adeseori codificate sub forma unor "macro-uri" sau a unor proceduri "în-line".
Pentru orice nod i, diferit de rădăcina, este adevărată următoarea *proprietate de heap*:
$A[\text{Parinte}(i)] \geq A[i]$
adică valoarea ataşata nodului este mai mică sau egală cu valoarea asociată părintelui său. Astfel cel mai mare element din heap este păstrat în rădăcină, iar valorile nodurilor oricărui subarbore al unui nod sunt mai mici sau egale cu valoarea nodului respectiv.
Definim *înălţimea* unui nod al arborelui ca fiind numărul muchiilor aparţinând celui mai lung drum care leagă nodul respectiv cu o frunză, iar înălţimea arborelui ca fiind înălţimea rădăcinii. Deoarece un heap având n elemente corespunde unui arbore binar complet, înălţimea acestuia este $\Theta(\log2\ n)$. Vom vedea că timpul de execuţie al operaţiilor de bază, care se efectuează pe un heap, este proporţional cu înălţimea arborelui şi este $\Theta(\log2\ n)$. În cele ce urmează, vom prezenta trei proceduri şi modul lor de utilizare în algoritmul de sortare, respectiv într-o structura de tip coada de prioritate.

- Procedura Reconstituie-Heap are timpul de execuţie Θ (log2 n) şi este de prima importanţa în întreţinerea proprietăţii de heap.
- Procedura Construieste-Heap are un timp de execuţie liniar şi generează un heap dintr-un vector neordonat, furnizat la intrare.
- Procedura Heapsort se executa în timpul $O(n \log2\ n)$ şi ordonează un vector în spaţiul alocat acestuia.

Procedura Reconstituie-Heap este un subprogram important în prelucrarea heap-urilor.
Datele de intrare ale acesteia sunt un vector A şi un indice i din vector. Atunci când se apelează Reconstituie-Heap, se presupune că subarborii, având ca rădăcini nodurile Stânga(i) respectiv Dreapta(i), sunt heap-uri. Dar, cum elementul $A[i]$ poate fi mai mic decât descendenţii săi, acesta nu respecta proprietatea de heap. Sarcina procedurii Reconstituie-Heap este

de a “scufunda” în heap valoarea *A*[*i*], astfel încât subarborele care are în rădăcina valoarea elementului de indice *i*, să devina un heap.

```
Subalgoritm Reconstituie-Heap(A, i)
l ← Stânga(i)
r ←Dreapta(i)
daca l ≤ dimesiune-heap[A] şi A[l] > A[i] atunci
      maxim ←l
altfel
      maxim ←i
daca r ≤ dimesiune-heap[A] şi A[r] > A[maxim] atunci
      maxim ←r
daca maxim <> i atunci
      schimba A[i] ↔ A[maxim]
      Reconstituie-Heap(A, maxim)
```

Figura 1.9

ilustrează efectul procedurii Reconstituie-Heap.

La fiecare pas se determina cel mai mare element dintre *A*[*i*], *A*[Stânga(*i*)] şi *A*[Dreapta(*i*)], iar indicele sau se păstrează în variabila *maxim*. Daca *A*[*i*] este cel mai mare, atunci subarborele având ca rădăcină nodul *i* este un heap şi procedura se termina. În caz contrar, cel mai mare element este unul dintre cei doi descendenţi şi *A*[*i*] este interschimbat cu *A*[*maxim*]. Astfel, nodul *i* şi descendenţii săi satisfac proprietatea de heap.
Nodul *maxim* are acum valoarea iniţiala a lui *A*[*i*], deci este posibil ca subarborele de rădăcină *maxim* să nu îndeplinească proprietatea de heap. Rezulta că procedura Reconstituie-Heap trebuie apelata recursiv din nou pentru acest subarbore.
Timpul de execuţie al procedurii Reconstituie-Heap, corespunzător unui arbore de rădăcină *i* şi dimensiune *n*, este $\Theta(1)$, timp în care se pot analiza relaţiile dintre *A*[*i*], *A*[Stânga(*i*)] şi *A*[Dreapta(*i*)] la care trebuie adăugat timpul în care Reconstituie-Heap se executa pentru subarborele având ca rădăcină unul dintre descendenţii lui *i*. Dimensiunea acestor subarbori este de cel mult $2n/3$ – cazul cel mai defavorabil fiind acela în care nivelul inferior al arborelui este plin exact pe jumătate – astfel, timpul de execuţie al procedurii Reconstituie-Heap poate fi descris prin următoarea inegalitate recursiva:
$T(n) \leq T(2n/3) + \Theta(1)$*:*
Timpul de execuţie al procedurii Reconstituie-Heap pentru un nod de înălţime *h* poate fi exprimat alternativ ca fiind egal cu $\Theta(h)$.

Figura 1.9 Efectul procedurii Reconstituie-Heap(*A,* 2),
unde *dimesiune-heap*[*A*] = 10. (a)
Configuraţia iniţiala a heap-ului, unde *A*[2] (pentru nodul *i* = 2), nu respecta proprietatea de heap deoarece nu este mai mare decât descendenţii săi. Proprietatea de heap este restabilita pentru nodul 2 în (b) prin interschimbarea lui *A*[2] cu *A*[4], ceea ce anulează proprietatea de heap pentru nodul 4. Apelul recursiv al procedurii Reconstituie-Heap(*A,* 4) poziţionează valoarea lui *i* pe 4. Dupa interschimbarea lui *A*[4] cu *A*[9], aşa cum se vede în (c), nodul 4 ajunge la locul sau şi apelul recursiv Reconstituie-Heap(*A,* 9) nu mai găseşte elemente care nu îndeplinesc proprietatea de heap.

Construirea unui heap

Procedura Reconstituie-Heap poate fi utilizata "de jos în sus" pentru transformarea vectorului *A*[1..*n*] în heap, unde *n* = *lungime*[*A*]. Deoarece toate elementele subşirului *A*[([*n*/2]+1)..*n*] sunt frunze, acestea pot fi considerate ca fiind heap-uri formate din câte un element. Astfel, procedura Construieste-Heap trebuie să traverseze doar restul elementelor şi să execute procedura Reconstituie-Heap pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor asigura că subarborii, având ca rădăcină descendenţi ai nodului *i* să formeze heap-uri înainte ca Reconstituie-Heap să fie executat pentru aceste noduri.

```
Subalgoritm Construieste-Heap(A)
1: dimesiune-heap[A] ← lungime[A]
2: pentru i ← [lungime[A]/2],1 executa
3:       Reconstituie-Heap(A, i)
```

Figura 1.10 ilustrează modul de funcţionare al procedurii Construieste-Heap.

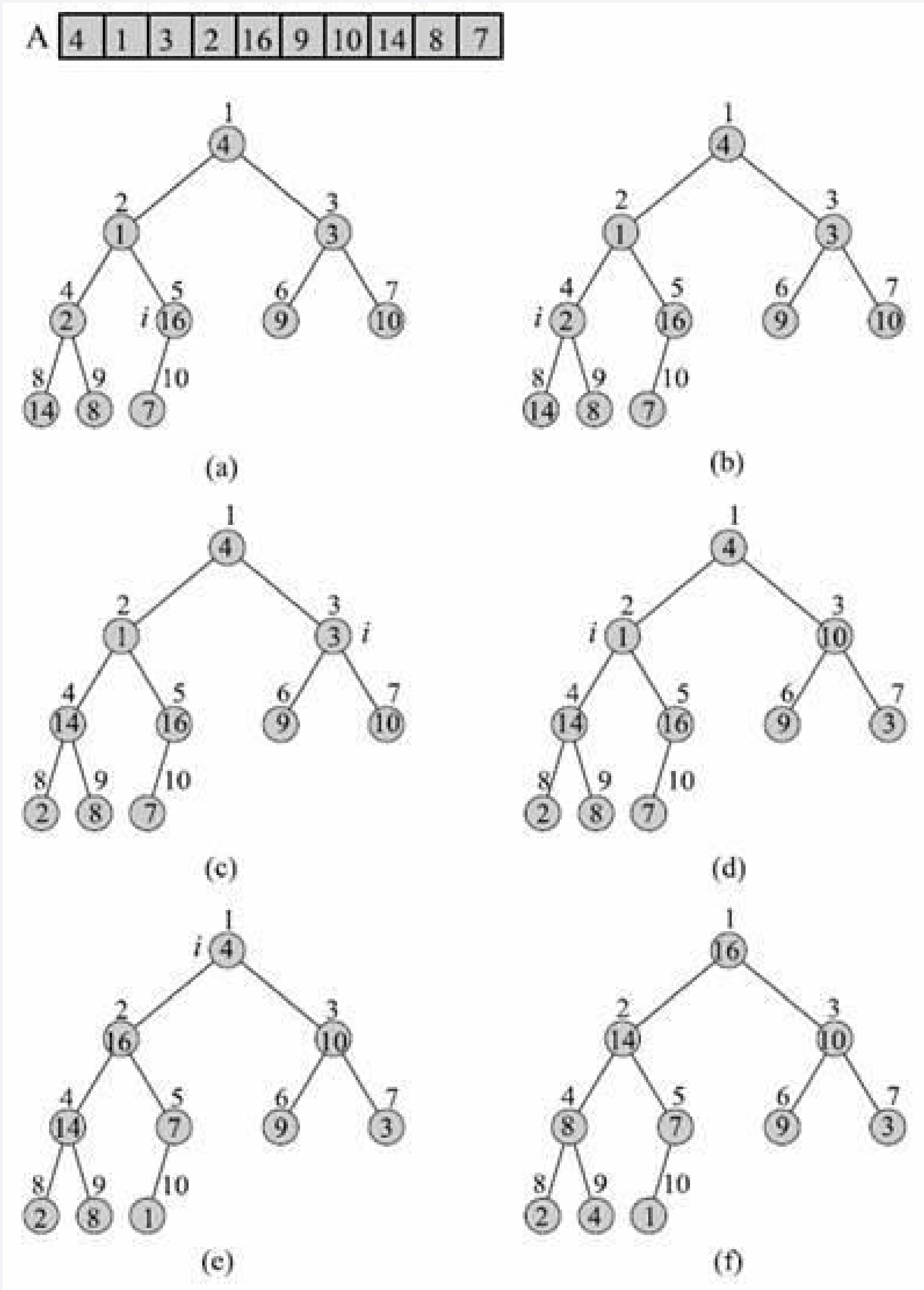


Figura 1.10

Modul de execuţie a procedurii Construieste-Heap. În figura se vizualizează structurile de date în starea lor anterioara apelului procedurii Reconstituie-Heap (linia 3 din procedura Construieste-Heap). (a) Se considera un vector *A* având 10 elemente şi arborele binar corespunzător. Dupa cum se vede în figura, variabila de control *i* a ciclului, în momentul apelului Reconstituie- Heap(*A, i*), indica nodul 5. (b) reprezintă rezultatul, variabila de control *i* a ciclului acum indica nodul 4. (c) - (e) vizualizează iteraţiile succesive ale ciclului pentru din Construieste-Heap. Se observa că, atunci când se apelează procedura Reconstituie-Heap pentru un nod dat, subarborii acestui nod sunt deja heap-uri. (f) reprezintă heap-ul final al procedurii Construieste-Heap.

Timpul de execuţie al procedurii Construieste-Heap poate fi calculat simplu, determinând limita superioara a acestuia: fiecare apel al procedurii Reconstituie-Heap necesita un timp Θ(log2 n) şi, deoarece pot fi Θ(n) asemenea apeluri, rezulta că timpul de execuţie poate fi cel mult Θ(n log2 n). Aceasta estimare este corecta, dar nu este suficient de tare asimptotic.
Vom obţine o limita mai tare observând că timpul de execuţie a procedurii Reconstituie-Heap depinde de înălţimea nodului în arbore, aceasta fiind mica pentru majoritatea nodurilor.
Estimarea noastră mai riguroasa se bazează pe faptul că un heap având n elemente are înălţimea log2n şi că pentru orice înălţime h, în heap exista cel mult $[n/2h+1]$noduri de înălţime h.
Timpul de execuţie a procedurii Reconstituie-Heap pentru un nod de înălţime h fiind Θ(h), obţinem pentru timpul de execuţie a procedurii Construieste-Heap:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Astfel, timpul de execuţie al procedurii Construieste-Heap poate fi estimat ca fiind:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

De aici rezulta că se poate construi un heap dintr-un vector într-un timp liniar.

Algoritmul heapsort
Algoritmul heapsort începe cu apelul procedurii Construieste-Heap în scopul transformării vectorului de intrare $A[1..n]$ în heap, unde n = *lungime*[A]. Deoarece cel mai mare element al vectorului este ataşat nodului rădăcină $A[1]$, acesta va ocupa locul definitiv în vectorul ordonat prin interschimbarea sa cu $A[n]$. În continuare, “excluzând” din heap cel de-al n-lea element (şi micşorând cu 1 *dimesiune-heap*[A]), restul de $A[1..(n - 1)]$ elemente se pot transforma uşor în heap, deoarece

subarborii nodului rădăcină au proprietatea de heap, cu eventuala excepţie a elementului ajuns în nodul rădăcină.

SubalgoritmHeapsort(*A*)
1: Construieste-Heap(*A*)
2: pentru $i \leftarrow$ *lungime*[*A*], 2 executa
3: schimba $A[1] \leftrightarrow A[i]$
4: *dimesiune-heap*[*A*] $\leftarrow$ *dimesiune-heap*[*A*] - 1
5: Reconstituie-Heap(*A,* 1)

Apelând procedura Reconstituie-Heap(*A,* 1) se restabileşte proprietatea de heap pentru vectorul $A[1..(n - 1)]$. Acest procedeu se repeta micşorând dimensiunea heap-ului de la n - 1 până la 2.

Figura 1.11 ilustrează, pe un exemplu, modul de funcţionare a procedurii Heapsort, dupa ce în prealabil datele au fost transformate în heap. Fiecare heap reprezintă starea iniţiala la începutul pasului iterativ (linia 2 din ciclul pentru).

Timpul de execuţie al procedurii Heapsort este $\Theta(n \log2\ n)$, deoarece procedura Construieste-Heap se executa într-un timp $\Theta(n)$, iar procedura Reconstituie-Heap, apelata de n-1 ori, se executa în timpul $\Theta(\log2\ n)$.

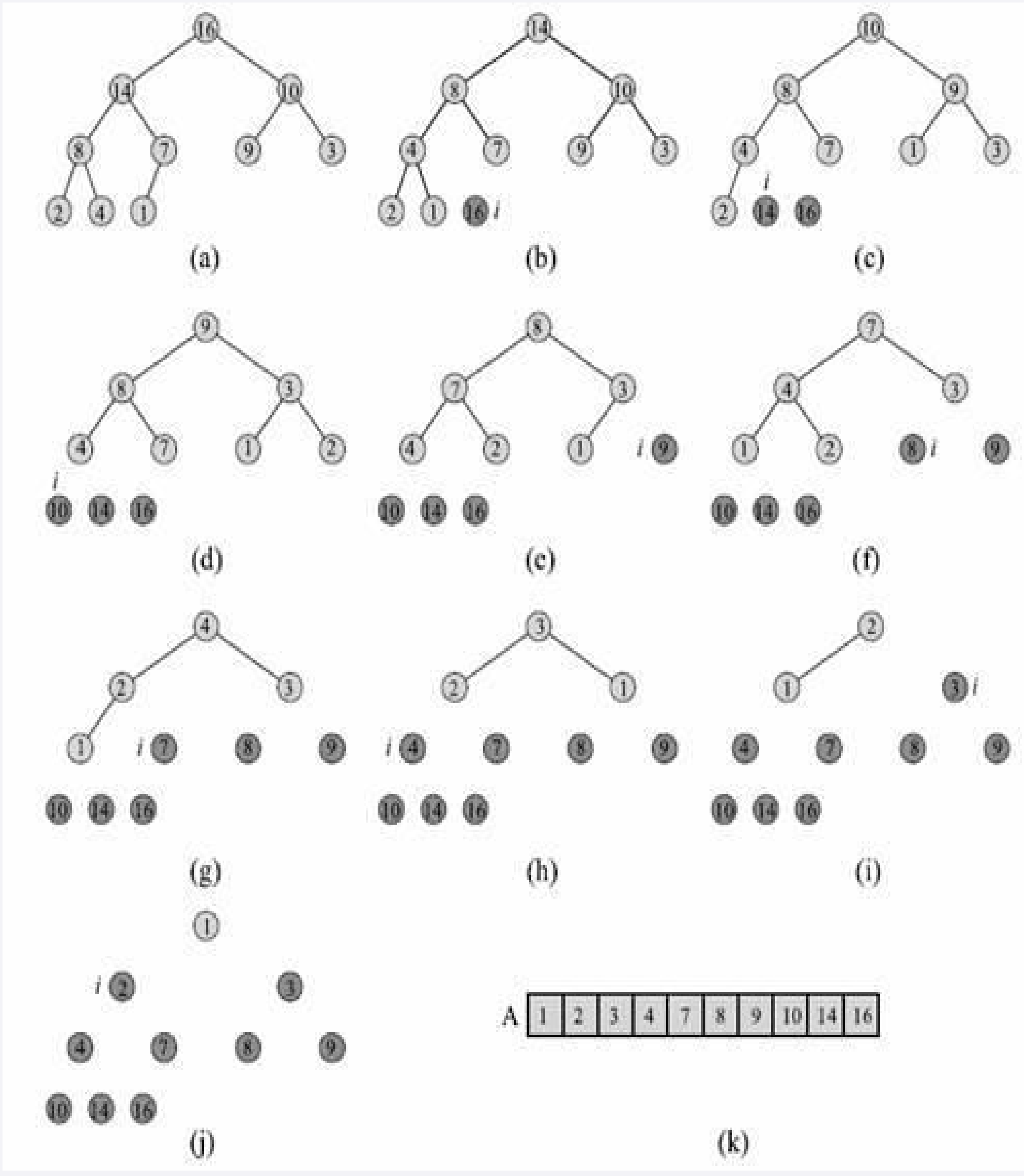


Figura 1.11 Modul de funcţionare a algoritmului Heapsort.

(a) Structura de date heap, imediat dupa construirea sa de către procedura Construieste-Heap.

(b)-(j) Heap-ul, imediat dupa câte un apel al procedurii Reconstituie-Heap (linia 5 în algoritm).

Figura reprezintă valoarea curenta a variabilei *i*. Din heap fac parte doar nodurile din cercurile nehaşurate. (k) Vectorul *A* ordonat, obţinut ca rezultat.

Fişa suport 7.8 Sortarea prin asamblare (heapsort)

Descriere

Se numeşte ansamblu (heap) a secvență de chei h1, h2,..., hn care satisfac condițiile: hi <= h2i si hi <= h2i+1 i=1,N/2.

Se aduce tabloul la forma unui ansamblu, adică pentru orice i,j, k din intervalul [1,N], unde j=2*i si k=2*i+1, să avem a[i]<=a[j] si a[i]<=a[k] (*). Se observă că în acest caz a[1] este elementul cu cheia minimă în tablou. Se interschimbă elementele a[1] şi a[N] şi se aduce subtabloul a[1],...,a[N-1] la forma de ansamblu, apoi se interschimbă elementele a[1] si a[N-1] şi se aduce subtabloul a[1],...,a[N-2] la forma de ansamblu ş.a.m.d. În final rezultă tabloul ordonat invers. Dacă se schimbă sensul relațiilor în condițiile (*) atunci se obține o ordonare directă a tabloului (a[1] va fi elementul cu cheia maximă).

Aducerea unui tablou la forma de ansamblu se bazează pe faptul că subtabloul a[N/2+1],...,a[N] este deja un ansamblu (nu există indicii j si k definiți ca mai sus). Acest subtablou se va extinde mereu spre stânga cu câte un element al tabloului, pâna când se ajunge la a[1]. Elementul adăugat va fi glisat astfel încât subtabloul extins să devină ansamblu.

Procedura Deplasare(s,d) realizează glisarea elementului a[s] astfel că subtabloul a[s],...,a[d] (s<d) să devină ansamblu. Această procedură este folosită mai întâi pentru aducerea întregului tablou la structura de ansamblu şi apoi pentru ordonarea tabloului conform metodei enunțate mai sus.

Timpul de execuție al sortării este O(N*log N).

Exemplu

Dorim să sortăm un şir de cinci valori de tip întreg:

Tablou: 9 1 7 0 3

Indici: 1 2 3 4 5

s=3 d=5 deplasare(2,5) rezultă tabloul: 9 3 7 0 1

s=1 d=5 deplasare(1,5) nu se efectuează deplasarea

s=1 d=4 deplasare(1,4) rezultă tabloul: 7 3 1 0 9

s=1 d=3 deplasare(1,3) rezultă tabloul: 3 0 1 7 9

s=1 d=2 deplasare(1,2) rezultă tabloul: 1 0 3 7 9

s=1 d=1 deplasare(1,1) rezultă tabloul: 0 1 3 7 9 – vector sortat

Algoritm descris în pseudocod

```
Deplasare(s,n)
i ← s    j ← 2*i    x ← a[i]  ok ← adevărat
cât timp j≤d şi ok≠0 execută
        dacă j<d atunci
                dacă a[j]<a[j+1] atunci
                j ← j+1
                sfârşit dacă


        sfârşit dacă
        dacă x< a[j] atunci
                a[i] ← a[j]  i ← j  j ← 2*i
                altfel  ok ← 1
        sfârşit dacă
sfârşit cât timp
a[i] ← x
Sfârşit subalgoritm
```

```
HeapSort
s ← [n/2]+1  d ← n
cât timp s>1 execută
        s ← s-1
        Apel Deplasare(s,n)
Sfârşit cât timp
Cât timp d>1 execută
         x ← a[1]   a[1] ← a[d]
  a[d] ← x   d ← d-1
  Apel Deplasare(s,n)
Sfârşit cât timp
Sfârşit subalgoritm
```